Pointer Analysis

JiangJun

Contents

Intro: What is it?

Alias Analysis:

When do two pointer expressions refers to the same storage location?

E.g.

```
{
    int x;
    p = &x;
    q = p;
}
```

$$\xleftarrow{}$$

$\boxed{(*p, *q), (x, *p), (x, *q)}$

Aliasing can arise due to

- Pointers

  …

- Call by reference

  e.g., void m(Object a, Object b){…}

  Let m(x, x) then [[a, b, x]]

  Let m(x, y) then [[a, x], [b, y]]

- Array indexing

  e.g., int i, j, a[100];

  Let i = j then [[a[i], a[j]]]

Why do we want to know?

Pointer analysis tells us what memory locations code uses or modifies and is useful in many analyses

E.g., Available expressions

*p = a + b; y = a + b;

If *p aliases a or b, then second computation of a+b is not redundent

E.g., Constant propagation

x = 3; *p = 4; y = x;

Is y constant? If *p and x do not alias each other, then yes. If *p and x always alias each other, then yes. If *p and x sometimes alias each other, then no.

The Pointer Alias Analysis Problem

Many uses:

Basic compiler optimizations

– register allocation, CSE, dead code elimination, live variables, instruction scheduling, loop invariant code motion, redundant load/store elimination

Parallelization

– instruction-level parallelism

– thread-level parallelism

Behavioral synthesis

– automatically converting C-code into gates

Error detection and program understanding

– memory leaks, wild pointers, security holes

The Pointer Alias Analysis Problem

Many challenges :

Complexity: huge in space and time

– compare every pointer with every other pointer

– at every program point

– potentially considering all program paths to that point

Scalability vs accuracy trade-off

– different analyses motivated for different purposes

– many useful algorithms(adds to confusion)

Coding corner cases

– pointer arithmetic(*p++), casting, function pointers, long-jump

– most algorithms require the entire program

– library code? optimizing at link-time only?

How ?
Some dimensions/
Design options

Intra-procedural / inter-procedural

Intra-procedural analysis

– a mechanism for performing optimization for each function in a compilation unit, using only the information available for that function and compilation unit.

Inter-procedural analysis

– a mechanism for performing optimization across function and compilation unit boundaries.

Flow-sensitive / flow-insensitive

Flow insensitive

- The order of statements doesn't matter
  - Result of analysis is the same regardless of statement order
- Uses a single global state to store results as they are computed
- Fast, but not very accurate

Flow sensitive

- The order of the statements matter
- Need a control flow graph
- Must store results for each program point
- Improves accuracy

Path sensitive

- Each path in a control flow graph is considered
- If-then-else implies mutually exclusive paths

Example

```
S1: a = malloc(…);
S2: b = malloc(…);
S3: a = b;
S4: a = malloc(…);
S5: if(c)
        a = b;
S6: if(!c)
        a = malloc(…);
S7: … = *a;
```

(assuming allocation-site heap modeling)

Flow Insensitive

aS7 —> {heapS1, heapS2, heapS4, heapS6}

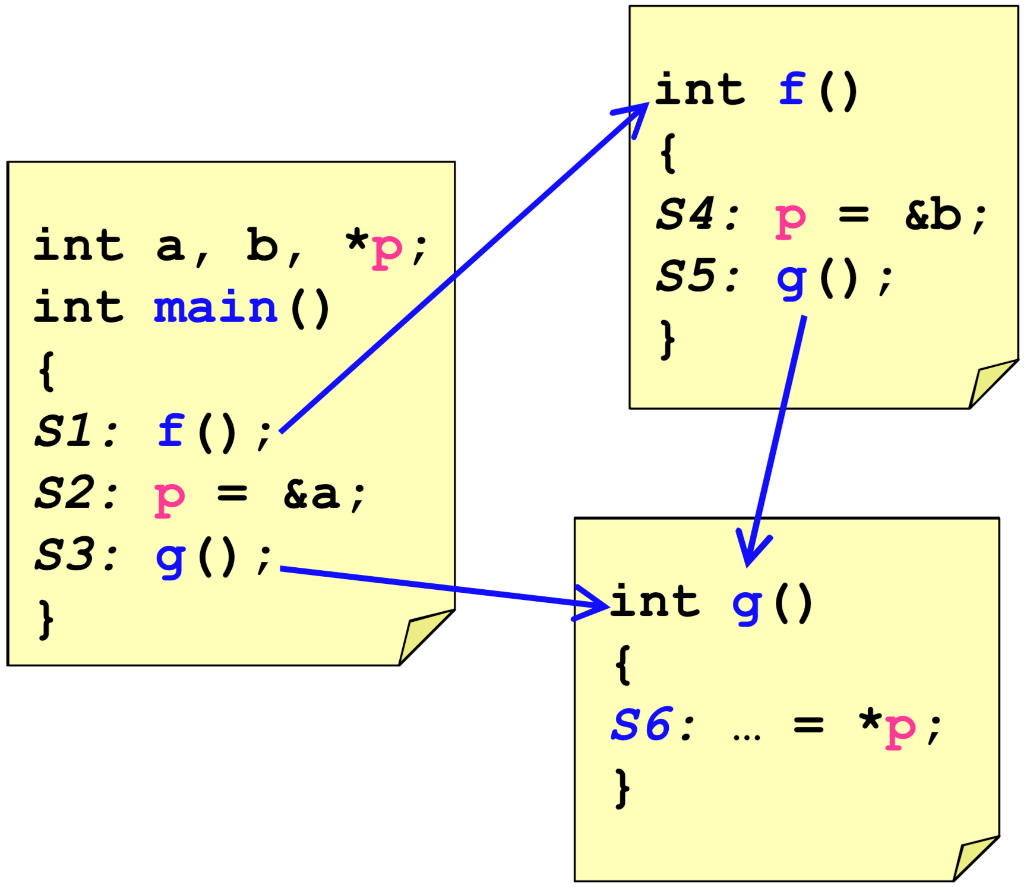
Flow Sensitive

aS7 —> {heapS2, heapS4, heapS6}

Path Sensitive

aS7 —> {heapS2, heapS6}

Context-sensitive / context-insensitive



```
int a, b, *p;
int main()
{
S1: f();
S2: p = &a;
S3: g();
}
```

```
int f()
{
S4: p = &b;
S5: g();
}
```

```
int g()
{
S6: … = *p;
}
```

whether to consider different calling contexts

e.g., what are the possibilities for p at S6?

Context Insensitive:

$p \Rightarrow \{a, b\}$

Context Sensitive:

Called from S3: $p \Rightarrow \{a\}$

Called from S5: $p \Rightarrow \{b\}$

Andersen’s Algorithm

Flow-insensitive, context-insensitive, iterative

Representation:

- one points-to graph for entire program
- each node represents exactly one location

For each statement, build the points-to graph:

`y = &x`	y points-to x
`y = x`	if x points-to w then y points-to w
`*y = x`	if y points-to z and x points-to w then z points-to w
`y = *x`	if x points-to z and z points-to w then y points-to w

Iterate until graph no longer changes

Worst case complexity: O(n3), where n = program size

```
p = &a
```

$$\begin{tikzcd}
p \arrow[r] & a
\end{tikzcd}$$

```
q = &b
```

$$\begin{tikzcd}
p \arrow[r] & a & \\
 & q \arrow[r] & b
\end{tikzcd}$$

```
*p = q;
```

$$\begin{tikzcd}
p \arrow[r] & a \arrow[dr] & \\
 & q \arrow[r] & b
\end{tikzcd}$$

```
r = &c;
```

$$\begin{tikzcd}
p \arrow[r] & a \arrow[dr] & & r \arrow[r] & c \\
 & q \arrow[r] & b & &
\end{tikzcd}$$

```
s = p;
```

$$\begin{tikzcd}
p \arrow[r] & a \arrow[ddr] & & r \arrow[r] & c \\
s \arrow[ur] & & & & \\
 & q \arrow[r] & b & &
\end{tikzcd}$$

```
t = *p;
```

$$\begin{tikzcd}
p \arrow[r] & a \arrow[ddr] & & r \arrow[r] & c \\
s \arrow[ur] & & & & \\
 & q \arrow[r] & b & & \\
t \arrow[urr] & & & &
\end{tikzcd}$$

```
*s = r;
```

$$\begin{tikzcd}
p \arrow[r] & a \arrow[ddr] \arrow[rrr, bend right] & & r \arrow[r] & c \\
s \arrow[ur] & & & & \\
 & q \arrow[r] & b & & \\
t \arrow[urr] & & & &
\end{tikzcd}$$

Pointer Analysis for Java

- Different languages use pointers differently
- *Scaling Java Points-To Anlaysis Using SPARK* Lhotak & Hendren CC 2003
  - Most C programs have many more occurrences of the address-of (&) operator than dynamic allocation
    - & creates stack-directed pointers; malloc creates heap-directed pointers
  - Java allows no stack-directed pointers, many more dynamic allocation sites than similar-sized C programs
  - Java strongly typed, limits set of objects a pointer can point to
    - Can improve precision
  - Call graph in Java depends on pointer analysis, and vice-versa (in context sensitive pointer analysis)
  - Dereference in Java only through field store and load
  - And more...
    - Larger libraries in Java, more entry points in Java, can't alias fields in Java, ...

Doop - Framework for Java Pointer and Taint Analysis

Doop only supports invocations from its home directory. The main options when running Doop are the analysis and the jar(s) options. For example, for a context-insensitive analysis on a jar file we issue:

```
$ ./doop -a context-insensitive -i com.example.some.jar
```

-a, --analysis: The name of the analysis.
-i, --input: The jar file(s) to analyse.
--platform: The platform and version to use. Doop checks for the JRE-specific or Android-specific files in the $DOOP_PLATFORMS_LIB directory. For instance, java_7 specifies the Java platform and that JRE 1.7 will be used to resolve library calls, while android_25_fulljars specifies the Android platform and the Android library version 25 will be used to resolve calls to the Android library).
--main: The name of the Java main class.
-t, --timeout: The analysis execution timeout in minutes.
-id, --identifier: The human-friendly identifier of the analysis (if not specified, Doop will generate one automatically).
--regex: The Java package names to analyse.
-p, --properties: Load options from the given properties file.

Thank you